МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсия

Студент гр. 8304 _____ Мухин А. М.
Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2019

Цель работы.

Получить опыт работы с бинарным деревом, реализовав его с помощью динамической памяти.

Задание.

Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов:

- определить максимальную глубину дерева *b*, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;
- вычислить длину внутреннего пути дерева *b*, т. е. сумму по всем узлам длин путей от корня до узла;
- напечатать элементы из всех листьев дерева *b*;
- подсчитать число узлов на заданном уровне *n* дерева *b* (корень считатькорень считать узлом 1-го уровня);

Вариант 2-д.

Описание алгоритма.

Все подзадачи решались с помощью итераторов, которые возвращали элементы по пути обхода в глубину или в ширину. Для возвращения следующего элемента был написан абстрактный класс TreeIterator, который имеет два виртуальных метода bool has_next(корень считать) и Node* next(корень считать), реализованные уже в классах наследниках TreeIteratorDFS и TreeIteratorBFS.

TreeIteratorDFS – класс, которые помимо унаследованных функций содержит такие поля, как unsigned int size – размер элементов массива, который заполняется по пути обхода в глубину в нужном порядке. std::vector<Node*> path_in_dip – этот массив, void go_in_dip(корень считатьNode*) - helper функция для конструктора.

TreeIteratorBFS - класс, которые помимо унаследованных функций содержит такие поля, как size (корень считатьтоже самое, что и в TreeIteratorDFS), std::map<Node*, bool> is_visit – словарь с пометками о том, заходили ли мы в

данный узел или нет, std::vector<Node*> path_in_breadth – та же функция, что и у std::vector<Node*> path_in_dip.

Tree – класс, содержащий в себе такие поля, как указатель на структуру Node, которая в свою очередь содержит поля int data, Node* left и Node* right (корень считатьданные узла и указатели на его левую и правую ветку соответственно), функцию void insert(корень считать) и её функцию helper, нужные для рекурсивной вставки элемента в дерево, std::map<Node*, int> get_nodes_and_levels(корень считать) - функция возвращающая словарь, состоящий из ссылки на элемент и номера его уровня, функции TreeIterator* make_iterator_DFS(корень считать) и TreeIterator* make_iterator_BFS(корень считать) возвращают новый объект соответствующего типа.

Выполнение работы.

Для реализации поставленной задачи были реализованы следующие методы в классе Tree: int dip(корень считать) - последовательно проходим итератором в ширину по всем элементам и вызывая функцию, которая по ссылке на узел выдаёт уровень на котором он лежит и сравниваем его с текущим, если он меньше, то переприсваиваем его и в конце концов возвращаем максимум, void print_leaves(корень считать) - также итератором(корень считатьлюбым) проходим по дереву и если слева и справа нет потомков, выводим этот элемент, int tree_length(корень считать) - вычисляет полную глубину дерева, как сымму количества элементов на данном уровне умноженную на (корень считатьэтот уровень – 1), int count_nodes_in_level(корень считатьint) – проходит по итератору в ширину и увеличивает значение счётчика, если переданный в функцию параметр равен значению по нынешнему ключу.

Разработанный программный код см. в приложении А.

Тестирование.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	15730311203510	Dip: 4 Leaf: 3

		Leaf: 10 Leaf: 20 Leaf: 35 Number of elements in the n level: 2 Dip of the tree: 13
2.	500 300 150 70 45 0 -9 -128 -939	Dip: 9 Leaf: -939 Number of elements in the n level: 1 Dip of the tree: 36
3.	1	Dip: 1 Leaf: 1 Number of elements in the n level: 0 Dip of the tree: 0

Выводы.

Ознакомились с основными понятиями и приёмами рекурсивного программирования, получили навыки программирования рекурсивных процедур и функций на языке программирования C++ и создания бинарного дерева, реализовав его с помощью динамической памяти. Научились работать и создавать итераторы для различного типа обходов дерева.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include "Tree.h"

int main(int argc, char* argv[])
    { if (argc < 2) {
           std::cout << "Set a root value!" << std::endl <<
std::string(40, '_') << std::string(3, '\n');
        return 1;
    } else {
```

```
        Tree tr(atoi(argv[1]));
        for (int i = 2; i < argc; i++) {
            tr.insert(atoi(argv[i]));
        }
        tr.print_tree();
        std::cout << "Dip: " <<tr.dip() << std::endl;
        tr.print_leaves();
         std::cout << "Number of elements in the n level: "<<
tr.count_nodes_in_level(2) << std::endl;
        std::cout << "Dip of the tree: "<< tr.tree_length() <<
std::endl << std::string(40, '_') << std::string(3, '\n');
        return 0;
    }
}
```

Название файла: Iterator.cpp

```
#include "Iterator.h"

TreeIteratorDFS::TreeIteratorDFS(Node* root) {
    if (root != nullptr) {
        path_in_dip.push_back(root);
        go_in_dip(root->left);
        go_in_dip(root->right);
    }
}

void TreeIteratorDFS::go_in_dip(Node* parent) {
    if (parent == nullptr) {
        return;
    } else {
        path_in_dip.push_back(parent);
        go_in_dip(parent->left);
        go_in_dip(parent->right);
    }
}

bool TreeIteratorDFS::has_next()
    { return size <=
    path_in_dip.size();
}

Node* TreeIteratorDFS::next() {
    return path_in_dip[size++];
}

TreeIteratorBFS::TreeIteratorBFS(Node* root) {
    if (root != nullptr) {
```

```
        std::queue<Node*> queue;
        queue.push(root);
        is_visit[root] = true;
        while(!queue.empty()) {
            Node* parent = queue.front();
            queue.pop();
            path_in_breadth.push_back(parent);
              if (parent->left != nullptr && !is_visit[parent-
>left]) {
                queue.push(parent->left);
                is_visit[parent->left] = true;
            }
             if (parent->right != nullptr && !is_visit[parent-
>right]) {
                queue.push(parent->right);
                is_visit[parent->right] = true;
            }
        }
    }
}

bool TreeIteratorBFS::has_next() {
    return size <= path_in_breadth.size();
}

Node* TreeIteratorBFS::next()
    { return path_in_breadth[size+
    +];
}
```

Название файла: Iterator.h

```
#ifndef ITERATOR
#define ITERATOR
#include "Tree.h"

class Node;

class TreeIterator {
public:
    virtual bool has_next() = 0;
    virtual Node* next() = 0;
};

class TreeIteratorDFS : public TreeIterator
    { private:
        unsigned int size = 0;
        std::vector<Node*> path_in_dip;
```

```
        void go_in_dip(Node*);
    public:
        explicit TreeIteratorDFS(Node* root);

        bool has_next() override;
        Node* next() override;

        ~TreeIteratorDFS() = default;
};

class TreeIteratorBFS : public TreeIterator
    { private:
        unsigned int size = 0;
        std::map<Node*, bool> is_visit;
        std::vector<Node*> path_in_breadth;
    public:
        explicit TreeIteratorBFS(Node* root);

        bool has_next() override ;
        Node* next() override;

        ~TreeIteratorBFS() = default;
};
#endif
```

Название файла: Tree.cpp

```
#include "Tree.h"
#include <iostream>

Tree::Tree(int root_data) {
    root = new Node;
    root->data = root_data;
}

TreeIterator* Tree::make_iterator_DFS() {
    return new TreeIteratorDFS(root);
}

TreeIterator* Tree::make_iterator_BFS() {
    return new TreeIteratorBFS(root);
}

void Tree::insert(int element) {
    if (root == nullptr) {
        root = new Node;
        root->data = element;
```

```
    } else {
        if (root->data > element) {
            insert(root->left, element);
        }

        if (root->data < element)
            { insert(root->right,
            element);
        }

        if (root->data == element) {
            root->data = element;
        }
    }
}

void Tree::insert(Node*& parent, int element) {
    if (parent == nullptr) {
        parent = new Node;
        parent->data = element;
    } else {
        if (parent->data > element)
            { insert(parent->left,
            element);
        }

        if (parent->data < element)
            { insert(parent->right,
            element);
        }

        if (parent->data == element) {
            parent->data = element;
        }
    }
}

void Tree::print_tree() {
               std::map<Node*, int> nodes_and_levels =
get_nodes_and_levels();
    TreeIterator* bfs = this-
    >make_iterator_BFS(); int level_of_dip;
     for (Node* el = bfs->next(); bfs->has_next(); el = bfs-
>next()) {
        level_of_dip = nodes_and_levels[el];
        std::cout << std::string(level_of_dip * 5, ' ') << el-
>data << std::endl;
```

```
}
```

```
}

int Tree::dip() {
    if (root == nullptr) {
        return 0;
    } else {
                    std::map<Node*, int> nodes_and_levels =
get_nodes_and_levels();
        TreeIterator* bfs = this->make_iterator_BFS();

        int max_level_of_dip = 0;

          for (Node *el = bfs->next(); bfs->has_next(); el =
bfs->next()) {
            if (max_level_of_dip < nodes_and_levels[el])
                { max_level_of_dip = nodes_and_levels[el];
            }
        }

        return max_level_of_dip;
    }
}

void Tree::print_leaves() {
    TreeIterator* dfs = this->make_iterator_DFS();
     for (Node* el = dfs->next(); dfs->has_next(); el = dfs-
>next()) {
        if (el->left == nullptr && el->right == nullptr)
            { std::cout << "Leaf: " << el->data <<
            std::endl;
        }
    }
}

int Tree::tree_length() {
                std::map<Node*, int> nodes_and_levels =
get_nodes_and_levels();
    TreeIterator* bfs = this->make_iterator_BFS();
    int max_level = 1;
    int level_of_dip;

     for (Node* el = bfs->next(); bfs->has_next(); el = bfs-
>next()) {
        level_of_dip = nodes_and_levels[el];
        if (max_level < level_of_dip) {
            max_level = level_of_dip;
        }
    }
```

```
    int length = 0;
    for (int i = 1; i <= max_level; i++) {
        length += count_nodes_in_level(i) * (i - 1);
    }

    return length;
}

int Tree::count_nodes_in_level(int data) {
    if (root == nullptr) {
        return 0;
    } else {
                    std::map<Node*, int> nodes_and_levels =
get_nodes_and_levels();
        TreeIterator* bfs = this->make_iterator_BFS();
        int level_of_dip;
        int count = 0;

          for (Node* el = bfs->next(); bfs->has_next(); el =
bfs->next()) {
            level_of_dip = nodes_and_levels[el];
            if (level_of_dip == data) {
                count++;
            }
        }

        return count;
    }
}

std::map<Node*, int> Tree::get_nodes_and_levels() {
    std::map<Node*, int> nodes_and_levels;
    if (root == nullptr) {
        std::cout << "Tree is empty" <<
        std::endl; nodes_and_levels[nullptr] = 0;
        return nodes_and_levels;
    } else {
        std::map<Node *, bool> is_visit;
        std::queue<std::pair<Node *, int>>
        queue; queue.push({root, 1});
        is_visit[root] = true;

        while (!queue.empty()) {
            auto [parent, level] = queue.front();

            nodes_and_levels[parent] = level;
```

```
            queue.pop();

              if (parent->left != nullptr && !is_visit[parent-
>left]) {
                queue.push({parent->left, level +
                1}); is_visit[parent->left] = true;
            }

             if (parent->right != nullptr && !is_visit[parent-
>right]) {
                queue.push({parent->right, level +
                1}); is_visit[parent->right] = true;
            }
        }

        return nodes_and_levels;
    }
}

void Tree::clear(Node* current_elem) {
    if (current_elem != nullptr) {
        clear(current_elem->left);
        clear(current_elem->right);
        delete current_elem;
    }
}

Tree::~Tree() {
    clear(root);
}
```

Название файла: Tree.h

```
#ifndef TREE
#define TREE
#include <iostream>
#include <map>
#include <queue>
#include "Iterator.h"

class TreeIterator;

struct Node{
    int data;
    Node* left = nullptr;
    Node* right = nullptr;
};
```

```
class Tree{
    private:
        Node* root;
        void insert(Node*&, int element);
        void clear(Node*);
        std::map<Node*, int> get_nodes_and_levels();
    public:
        explicit Tree(int);

        TreeIterator* make_iterator_DFS();
        TreeIterator* make_iterator_BFS();

        void insert(int element);
        void print_tree();

        int dip();
        void print_leaves();
        int tree_length();
        int count_nodes_in_level(int);

        ~Tree();
};

#endif
```